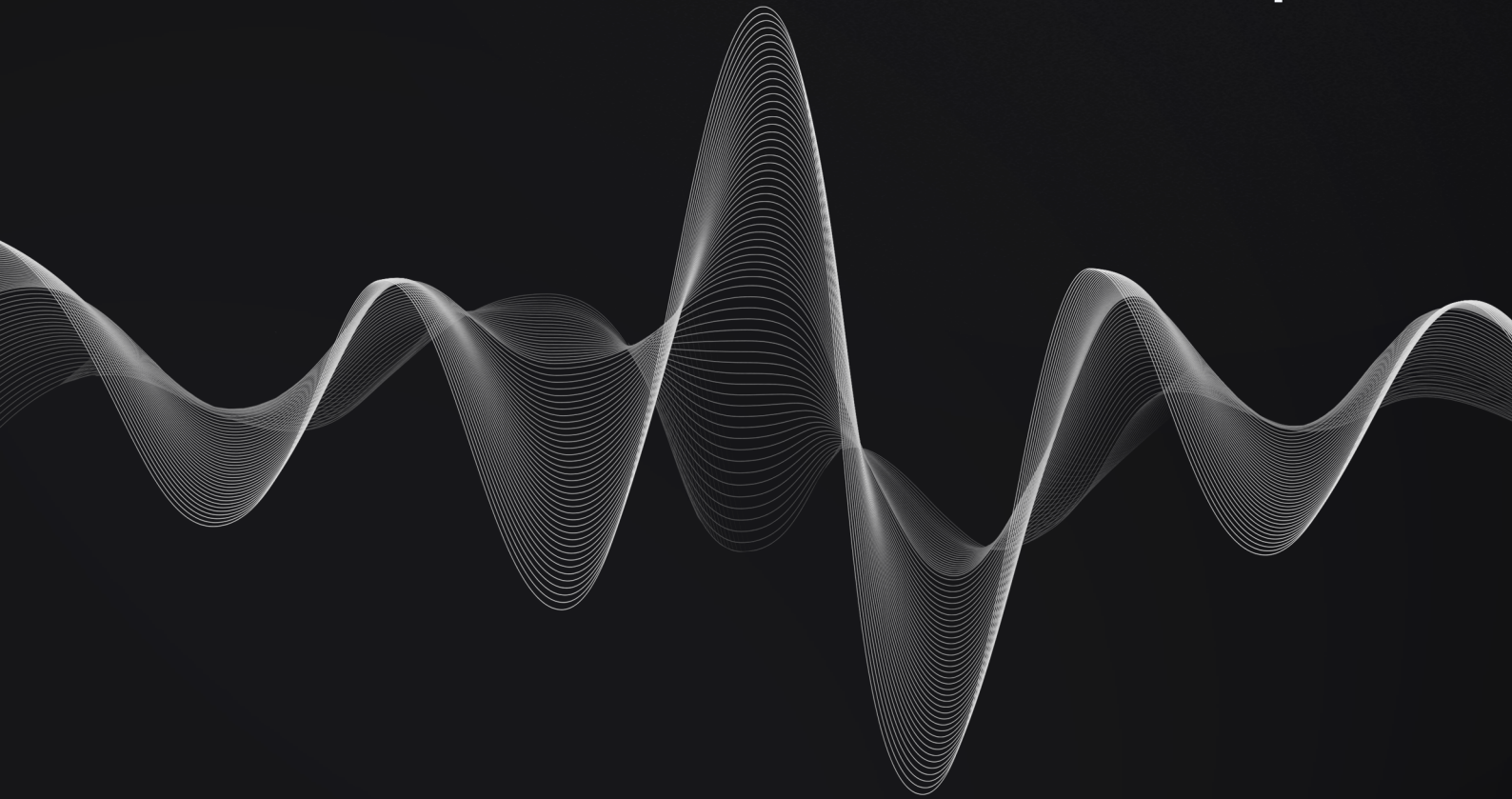




**Cube3**

# Cube3 Protocol RASP v2 Audit Report










# Document Control

**PUBLIC**

**FINAL**(v2.1)

## Audit\_Report\_CUBE-RSP\_FINAL\_21

Feb 19, 2024		v0.1	João Simões: Initial draft
Feb 22, 2024		v0.2	Michał Bajor: Added findings
Feb 22, 2024		v0.3	João Simões: Added findings
Feb 23, 2024		v1.0	Charles Dray: Approved
Mar 26, 2024		v1.1	João Simões: Reviewed findings
Apr 12, 2024		v2.0	Charles Dray: Finalized
Aug 14, 2024		v2.1	Charles Dray: Published

### Points of Contact

Craig Pickard  
Charles Dray

Cube3  
Resonance

craigp@cube3.ai  
charles@resonance.security

### Testing Team

Michał Bajor  
João Simões  
Michał Bazyli  
Ilan Abitbol

Resonance  
Resonance  
Resonance  
Resonance

michal.bajor@resonance.security  
joao.simoes@resonance.security  
michal.bazyli@resonance.security  
ilan.abitbol@resonance.security

## Copyright and Disclaimer

© 2024 Resonance Security, Inc. All rights reserved.

The information in this report is considered confidential and proprietary by Resonance and is licensed to the recipient solely under the terms of the project statement of work. Reproduction or distribution, in whole or in part, is strictly prohibited without the express written permission of Resonance.

All activities performed by Resonance in connection with this project were carried out in accordance with the project statement of work and agreed-upon project plan. It's important to note that security assessments are time-limited and may depend on information provided by the client, its affiliates, or partners. As such, the findings documented in this report should not be considered a comprehensive list of all security issues, flaws, or defects in the target system or codebase.

Furthermore, it is hereby assumed that all of the risks in electing not to remedy the security issues identified henceforth are sole responsibility of the respective client. The acknowledgement and understanding of the risks which may arise due to failure to remedy the described security issues, waives and releases any claims against Resonance, now known or hereafter known, on account of damage or financial loss.

# Contents

<b>1 Document Control</b>	<b>2</b>
Copyright and Disclaimer .....	2
<b>2 Executive Summary</b>	<b>4</b>
System Overview .....	4
Repository Coverage and Quality.....	4
<b>3 Target</b>	<b>6</b>
<b>4 Methodology</b>	<b>7</b>
Severity Rating.....	8
Repository Coverage and Quality Rating.....	9
<b>5 Findings</b>	<b>10</b>
Missing getIsProtocolPaused() Validation .....	11
Possibility Of Bypassing Revoked Integration Status Using Registered Integration .....	12
Floating Pragma On Core Protocol .....	13
No Usage of OpenZeppelin's PausableUpgradeable Contract.....	14
Missing ERC165 Validation Via supportsInterface() .....	15
Redundant Data Validation Of moduleAddress .....	16
<b>A Proof of Concepts</b>	<b>17</b>

# Executive Summary

**Cube3** contracted the services of Resonance to conduct a comprehensive security audit of their smart contracts between February 12, 2024 and February 23, 2024. The primary objective of the assessment was to identify any potential security vulnerabilities and ensure the correct functioning of smart contract operations.

During the engagement, Resonance allocated 2 main engineers to perform the security review. The engineers, including an accomplished professional with extensive proficiency in blockchain and smart-contract security, encompassing specialized skills in advanced penetration testing, and in-depth knowledge of multiple blockchain protocols, devoted 10 days to the project. The project's test targets, overview, and coverage details are available throughout the next sections of the report.

The ultimate goal of the audit was to provide Cube3 with a detailed summary of the findings, including any identified vulnerabilities, and recommendations to mitigate any discovered risks. The results of the audit are presented in detail further below.



## System Overview

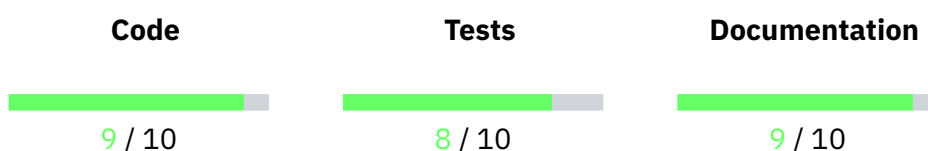
The Cube3 Protocol is a security tool that implements RASP (Runtime Application Self-Protection) functionality for on-chain traffic. Web3 applications and smart contracts are protected from malicious activities through monitoring and filtering of incoming traffic, acting to some extent as a Web3 Application Firewall. Cube3's protocol has been the target of a major code overhaul, which resulted in a modernized, simpler, and more secure version of the protocol.

The system can be deployed on multiple Ethereum-based chains and is best used with Cube3's Risk API. It essentially comprises 4 important components, the customer's integration registered to be protected by Cube3, the router used to forward protected traffic, the modules that different security functionality, and the on-chain registry which interfaces with Cube3's Key Management System. The components are expected to be protected with required access control mechanisms and to communicate securely between themselves.

As a general workflow, traffic sent to a customer's registered application or smart contract is forwarded by the router to the specified module. This is accomplished with the usage of a secure payload retrieved off-chain, and then exchanged throughout the components of the system. At the end, verified payloads are eventually capable of validating or invalidating on-chain activities of customers users.



## Repository Coverage and Quality



Resonance's testing team has assessed the Code, Tests, and Documentation coverage and quality of the system and achieved the following results:

- The code follows development best practices and makes use of known patterns, standard libraries, and language guides. It is easily readable and uses the latest stable version of relevant components. Overall, **code quality is excellent**.
- Unit and integration tests are included. The tests cover both technical and functional requirements. Code coverage is 100%. Overall, **tests coverage and quality is excellent..**
- The documentation includes the specification of the system, technical details for the code, relevant explanations of workflows and interactions. Overall, **documentation coverage and quality is excellent..**

# Target

The objective of this project is to conduct a comprehensive review and security analysis of the smart contracts that are contained within the specified repository.

The following items are included as targets of the security assessment:

- Repository: [cube-web3/protocol-core-solidity/src](#)
- Hash: 285d4972e4b80a10394aa7c2c1bdd02caf352f55
- Repository: [cube-web3/protection-solidity/src](#)
- Hash: f4ee8beafd718c93e03022c8e93eeeb1da5269e1

The following items are excluded:

- External and standard libraries
- Files pertaining to the deployment process
- Financial-related attack vectors

# Methodology

In the context of security audits, Resonance's primary objective is to portray the workflow of a real-world cyber attack against an entity or organization, and document in a report the findings, vulnerabilities, and techniques used by malicious actors. While several approaches can be taken into consideration during the assessment, Resonance's core value comes from the ability to correlate automated and manual analysis of system components and reach a comprehensive understanding and awareness with the customer on security-related issues.

Resonance implements several and extensive verifications based off industry's standards, such as, identification and exploitation of security vulnerabilities both public and proprietary, static and dynamic testing of relevant workflows, adherence and knowledge of security best practices, assurance of system specifications and requirements, and more. Resonance's approach is therefore consistent, credible and essential, for customers to maintain a low degree of risk exposure.

Ultimately, product owners are able to analyze the audit from the perspective of a malicious actor and distinguish where, how, and why security gaps exist in their assets, and mitigate them in a timely fashion.

## Source Code Review - Solidity EVM

During source code reviews for Web3 assets, Resonance includes a specific methodology that better attempts to effectively test the system in check:

1. Review specifications, documentation, and functionalities
2. Assert functionalities work as intended and specified
3. Deploy system in test environment and execute deployment processes and tests
4. Perform automated code review with public and proprietary tools
5. Perform manual code review with several experienced engineers
6. Attempt to discover and exploit security-related findings
7. Examine code quality and adherence to development and security best practices
8. Specify concise recommendations and action items
9. Revise mitigating efforts and validate the security of the system

Additionally and specifically for Solidity EVM audits, the following attack scenarios and tests are recreated by Resonance to guarantee the most thorough coverage of the codebase:

- Reentrancy attacks
- Frontrunning attacks
- Unsafe external calls
- Unsafe third party integrations
- Denial of service
- Access control issues



- Inaccurate business logic implementations
- Incorrect gas usage
- Arithmetic issues
- Unsafe callbacks
- Timestamp dependence
- Mishandled panics, errors and exceptions



## Severity Rating

Security findings identified by Resonance are rated based on a Severity Rating which is, in turn, calculated off the **impact** and **likelihood** of a related security incident taking place. This rating provides a way to capture the principal characteristics of a finding in these two categories and produce a score reflecting its severity. The score can then be translated into a qualitative representation to help customers properly assess and prioritize their vulnerability management processes.

The **impact** of a finding can be categorized in the following levels:

1. Weak - Inconsequential or minimal damage or loss
2. Medium - Temporary or partial damage or loss
3. Strong - Significant or unrecoverable damage or loss

The **likelihood** of a finding can be categorized in the following levels:

1. Unlikely - Requires substantial knowledge or effort or uncontrollable conditions
2. Likely - Requires technical knowledge or no special conditions
3. Very Likely - Requires trivial knowledge or effort or no conditions

		Likelihood		
		Very Likely	Likely	Unlikely
Impact	Strong	Critical	High	Medium
	Medium	High	Medium	Low
	Weak	Medium	Low	Info





# Repository Coverage and Quality Rating

The assessment of Code, Tests, and Documentation coverage and quality is one of many goals of Resonance to maintain a high-level of accountability and excellence in building the Web3 industry. In Resonance it is believed to be paramount that builders start off with a good supporting base, not only development-wise, but also with the different security aspects in mind. A product, well thought out and built right from the start, is inherently a more secure product, and has the potential to be a game-changer for Web3's new generation of blockchains, smart contracts, and dApps.

Accordingly, Resonance implements the evaluation of the code, the tests, and the documentation on a score **from 1 to 10** (1 being the lowest and 10 being the highest) to assess their quality and coverage. In more detail:

- Code should follow development best practices, including usage of known patterns, standard libraries, and language guides. It should be easily readable throughout its structure, completed with relevant comments, and make use of the latest stable version components, which most of the times are naturally more secure.
- Tests should always be included to assess both technical and functional requirements of the system. Unit testing alone does not provide sufficient knowledge about the correct functioning of the code. Integration tests are often where most security issues are found, and should always be included. Furthermore, the tests should cover the entirety of the codebase, making sure no line of code is left unchecked.
- Documentation should provide sufficient knowledge for the users of the system. It is useful for developers and power-users to understand the technical and specification details behind each section of the code, as well as, regular users who need to discern the different functional workflows to interact with the system.

# Findings

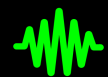
During the security audit, several findings were identified to possess a certain degree of security-related weaknesses. These findings, represented by unique IDs, are detailed in this section with relevant information including Severity, Category, Status, Code Section, Description, and Recommendation. Further extensive information may be included in corresponding appendices should it be required.

An overview of all the identified findings is outlined in the table below, where they are sorted by Severity and include a **Remediation Priority** metric asserted by Resonance's Testing Team. This metric characterizes findings as follows:

- ..... **"Quick Win"** Requires little work for a high impact on risk reduction.
- ....|.. **"Standard Fix"** Requires an average amount of work to fully reduce the risk.
- ...||| **"Heavy Project"** Requires extensive work for a low impact on risk reduction.

---

<b>RES-01</b>	Missing getIsProtocolPaused() Validation	.....	Resolved
<b>RES-02</b>	Possibility Of Bypassing Revoked Integration Status Using Registered Integration	.... ..	Acknowledged
<b>RES-03</b>	Floating Pragma On Core Protocol	.....	Resolved
<b>RES-04</b>	No Usage of OpenZeppelin's PausableUpgradeable Contract	.....	Acknowledged
<b>RES-05</b>	Missing ERC165 Validation Via supportsInterface()	.....	Acknowledged
<b>RES-06</b>	Redundant Data Validation Of moduleAddress	.....	Resolved



# Missing `getIsProtocolPaused()` Validation

Low

RES-CUBE-RSP01

Data Validation

Resolved

## Code Section

- `protocol-core-solidity/src/abstracts/IntegrationManagement.sol#L49`
- `protocol-core-solidity/src/abstracts/IntegrationManagement.sol#L60`
- `protocol-core-solidity/src/abstracts/IntegrationManagement.sol#L66`
- `protocol-core-solidity/src/abstracts/IntegrationManagement.sol#L105`

## Description

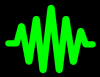
The functions `transferIntegrationAdmin()`, `acceptIntegrationAdmin()`, `updateFunctionProtectionStatus()`, and `initiateIntegrationRegistration()` are capable of modifying the storage of the Router smart contract, yet they do not validate whether the protocol is paused via the function `getIsProtocolPaused()`.

## Recommendation

It is recommended to properly validate and ensure that changes made to the smart contract's storage can only happen when the protocol is not paused due to an emergency.

## Status

*The issue has been fixed in `37a0f9c1bbd1c2e96559d4f62cda6d4e5bbb81aa`.*



# Possibility Of Bypassing Revoked Integration Status Using Registered Integration

Low

RES-CUBE-RSP02

Business Logic

Acknowledged

## Code Section

- [protocol-core-solidity/src/Cube3RouterImpl.sol#L98-L139](#)

## Description

It was observed that in certain scenarios it is possible to bypass one REVOKED integration status via another REGISTERED one. The scenario is as follows:

1. Suppose there are two integrations connected to the protocol, one of them is REGISTERED and another one is REVOKED.
2. The malicious owner of a REVOKED integration colludes with the corruptible owner of a REGISTERED integration.
3. The REGISTERED integration calls the function `routeToModule` directly, however using the arguments received from the REVOKED integration.
4. This results in a REGISTERED integration using the Cube3 protocol for the REVOKED one's benefit effectively bypassing the REVOKED status.

It should be noted that any future modules that do not make use of `msg.sender` will be more impacted by this finding.

## Recommendation

While there is no full-proof resolution for this issue, it is recommended to validate that the `integrationFnCallSelector` received in `integrationCallData` indeed exists and belongs to the integration identified by `msg.sender`. One possible solution to this would be to implement ERC165 for all registered and revoked integrations.

## Status

*The issue was acknowledged by Cube3's team. The development team stated "At this time, having one integration access the protocol via another is not deemed a security risk, albeit a violation of CUBE3's terms. Registering an integration's functions with the protocol does not mitigate this issue, as the colluding (registered) integration can construct the `msgData` passed into `routeToModule` and add any 4 byte selector to the front of the byte data. Because of this, should they wish, the colluding integration could simply register this new function signature with the protocol and enable/disable protection as they see fit. ERC165 is easily spoofed as a method of validation and adds additional gas to the call. Should additional modules be added in the future that do not require the validation of `msg.sender`, this approach can be revisited and a change made through a protocol upgrade."*



# Floating Pragma On Core Protocol

Info

RES-CUBE-RSP03

Code Quality

Resolved

## Code Section

Not specified.

## Description

Floating pragmas is a feature of Solidity that allows developers to specify a range of compiler versions that can be used to compile the smart contract code. For security purposes specifically, the usage of floating pragmas is discouraged and not recommended. Contracts should be compiled and deployed with a strict pragma, the one which was thoroughly tested the most by developers. Locking the pragma helps ensure that contracts do not accidentally get deployed using too old or too recent compiler versions that might introduce security issues that impact the contract negatively.

It should be noted however that, pragma statements can be allowed to float when a contract is intended for consumption by other developers, as in the case with contracts in a library or EthPM package.

The smart contracts of Cube3's core protocol make use of floating pragmas. These are not intended for use as libraries for other developers and, as such, should be locked of their pragma.

## Recommendation

It is recommended to lock the pragma of all smart contracts to the same Solidity compiler version.

## Status

*The issue has been fixed in 37a0f9c1bbd1c2e96559d4f62cda6d4e5bbb81aa.*



# No Usage of OpenZeppelin's PausableUpgradeable Contract

Info

RES-CUBE-RSP04

Code Quality

Acknowledged

## Code Section

- [protocol-core-solidity/src/abstracts/RouterStorage.sol](#)
- [protocol-core-solidity/src/common/Structs.sol#L52](#)

## Description

OpenZeppelin implements several standards used all across blockchain development, especially in Solidity and the Ethereum Virtual Machine. These standards primary goal is to unify and normalize development patterns so that the entire blockchain may be more understandable and readily usable.

The Cube3 smart contracts implement source code that mimics or is very similar to components already implemented by OpenZeppelin, and could therefore, be switched with the more the standard well-known approach. Such source code is included in:

- RouterStorage.sol contract pausability features, can be substituted with OpenZeppelin's implementation of PausableUpgradeable.

It should be noted that the usage of bigger libraries does not constitute an overuse of gas, since only the functions that are used are included on the bytecode of the smart contract.

## Recommendation

It is recommended to make use of implemented and audited standards that already solve the necessary functionalities.

## Status

*The issue was acknowledged by Cube3's team. The development team stated "The custom pause functionality is implemented with gas efficiency in mind by using a single storage slot to store the pause status along with the registry, reducing SLOADS and warming the storage slot for future use in the transaction lifecycle."*



# Missing ERC165 Validation Via supportsInterface()

Info

RES-CUBE-RSP05

Data Validation

Acknowledged

## Code Section

- [protocol-core-solidity/src/modules/SecurityModuleBase.sol#L38](#)
- [protocol-core-solidity/src/Cube3RouterImpl.sol#L66-L70](#)
- [protocol-core-solidity/src/abstracts/ProtocolManagement.sol#L83](#)
- [protocol-core-solidity/src/ProtectionBase.sol#L113](#)

## Description

In several places across the codebase, contracts reference addresses of other contracts. A validation is performed that simply verifies whether these addresses are equal to the zero-address. This assures calling only an existing address in the blockchain, however, it does not ensure that this address is able to properly receive the call. Such validation is performed via ERC165's `supportsInterface()` function which makes sure that a given address implements an expected interface. Additionally, such a validation indirectly asserts that the given address is not a zero-address.

## Recommendation

It is recommended to implement an ERC165 based validation and utilize `supportsInterface()` function to assure that a given address implements expected functionality and is not a zero-address.

## Status

*The issue was acknowledged by Cube3's team. The development team stated "The protocol contracts are all deployed by the CUBE3 team and off-chain validation, as well as fork tests, will be used to validate addresses. Should an incorrect contract be deployed, it can simply be updated via mechanisms available in the protocol. As far as Protection-Base.sol, adding ERC165Checker adds an external dependency and increases contract size, both of which are undesirable. The use of supportsInterface would ensure that the contract address passed as router is the correct address pointing to the CUBE3 Router. This functionality is achieved by the new `_assertPreRegistrationSucceeds` function used in `_baseInitProtection()`."*





# Redundant Data Validation Of moduleAddress

Info

RES-CUBE-RSP06

Gas Optimization

Resolved

## Code Section

- [protocol-core-solidity/src/abstracts/ProtocolManagement.sol#L83](#)

## Description

The ProtocolManagement contract defines an `installModule()` function that saves new module's address into the storage. It was observed that it performs a zero-address check that is then followed by ERC165's `supportsInterface()` verification. Because the ERC165 standard is used, the zero-address check is redundant as the `supportsInterface()` function will fail if called on the zero-address. Effectively, the zero-address check is performed twice which incurs a higher gas costs.

## Recommendation

It is recommended to remove the redundant explicit zero-address validation and leave only the ERC165 based `supportsInterface()` validation.

## Status

*The issue has been fixed in 37a0f9c1bbd1c2e96559d4f62cda6d4e5bbb81aa.*

# Proof of Concepts

## RES-02 Possibility Of Bypassing Revoked Integration Status Using Registered Integration

*DemoRevoked.sol:*

```
// SPDX-License-Identifier: MIT
pragma solidity >= 0.8.19 < 0.8.24;

import { ICube3RouterMinimal } from "@cube3/interfaces/ICube3RouterMinimal.sol";
import { Cube3Protection } from "@cube3/Cube3Protection.sol";

contract DemoRevoked is Cube3Protection {
    event Success();

    constructor(address _router) Cube3Protection(_router, msg.sender, true) {}

    function oldNoArgsFunction(bytes calldata cubePayload) public
    ↪ cube3Protected(cubePayload) {
        emit Success();
    }

    function newNoArgsFunction(address payable _friendIntegration, bytes calldata
    ↪ cubePayload) public payable {
        (bool success, ) = _friendIntegration.call(
    ↪ abi.encodeWithSignature("forwardFromRevokedToRouter(address,uint256,bytes)",
    ↪ msg.sender, msg.value, msg.data)
        );

        require(success, "Failed");
    }
}
```

*DemoRegistered.sol:*

```
// SPDX-License-Identifier: MIT
pragma solidity >= 0.8.19 < 0.8.24;

import { ICube3RouterMinimal } from "@cube3/interfaces/ICube3RouterMinimal.sol";
import { Cube3Protection } from "@cube3/Cube3Protection.sol";

contract Demo is Cube3Protection {
    constructor(address _router) Cube3Protection(_router, msg.sender, true) {}

    function forwardFromRevokedToRouter(
        address payable msgSender,
        uint256 msgValue,
        bytes calldata msgData
    )
```

```

    ) public payable {
        // Copy and paste from _assertShouldProceedAndCall()
        try ICube3RouterMinimal(_cube3Storage().router).routeToModule(msgSender,
↪ msgValue, msgData) returns (
            bytes32 result
        ) {
            // Checks: the call succeeded with the expected return value.
            if (result != keccak256("CUBE3_PROCEED_WITH_CALL")) {
                revert Cube3Protection_InvalidRouterReturn();
            }
            return;
        } catch (bytes memory revertData) {
            // Bubble up the revert data to capture the original error from the
↪ protocol.
            assembly {
                revert(add(revertData, 0x20), mload(revertData))
            }
        }
    }
}

```